Scribe Notes for *Algorithmic Number Theory*
Class 5—May 22, 1998

Scribes: Cara Struble and Craig Struble

Abstract

Today we introduce complexity theory. We show how a problem can be represented by a language, and we define the complexity classes P and NP.

1 Complexity Theory

Complexity theory is the study of problems and algorithms to solve those problems. The goal is to classify problems into levels of difficulty. When problems are classified, it is possible to know which problems are **tractable**, or computable using an acceptable amount of resources on a computer, or **intractable**, those that cannot be computed using reasonable resources. To study a given problem and its difficulty, three things are needed:

- a problem statement,
- a **model of computation**, that is, a description of a computing machine,
- and an algorithm to solve the problem for a given model of computation.

Figure 1 displays the connections between problems, models of computation, and algorithms to obtain an understanding of the complexity for a problem.

In complexity theory, the term **instance** means the input to a problem. An algorithm takes an instance and computes a solution using a given model of computation. The complexity of a problem, obtained through analysis of algorithms to solve the problem, may focus on different resources used by the algorithms. In this class, we are primarily concerned with **time complexity**, the number of "steps" an algorithm requires to compute a solution. Also of interest is **space complexity**, the number of "memory locations" used by an algorithm to compute a solution.

1.1 Input Size

The complexity of an algorithm is analyzed by comparing how algorithms use resources asymptotically. Usually, the complexity is in terms of **worst case complexity**, the maximum amount of resources used by an algorithm asymptotically. The complexity is a function of the size of the representation of an instance.

For the problems in this class, an instance will usually consist of numbers. The size of an instance reflects the number of bits needed to represent the number. The text (pg. 41) designates $\lg n$ to be the number of bits in the binary representation of the integer n. The function $\lg n$ is defined as follows:

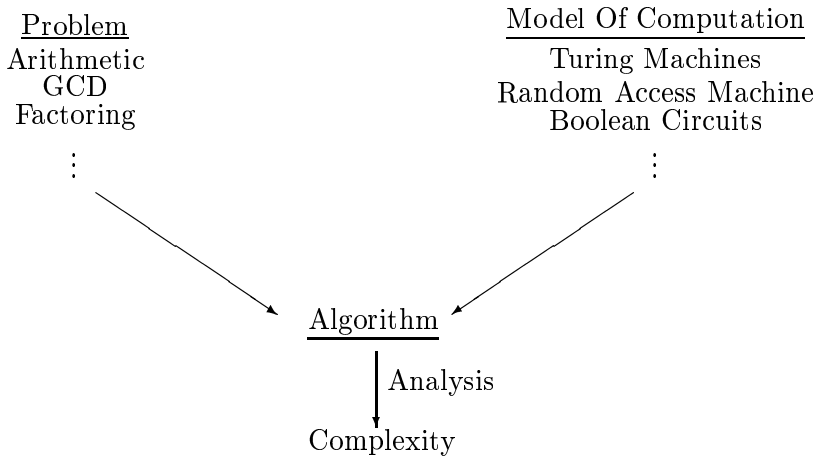

Figure 1: Relations between problems, models of computation, and algorithms.

Operation	Naive Complexity
$a \pm b$	$\lg a + \lg b$
$a * b$	$(\lg a)(\lg b)$
$a = qb + r$	$(\lg q)(\lg b)$

Table 1: (Table 3.1) Cost of arithmetic operations

$$\lg n \quad = \quad \begin{cases} 1, & \text{if } n = 0; \\ 1 + \lfloor \log_2 n \rfloor, & \text{if } n \neq 0. \end{cases}$$

A step in an algorithm corresponds to one bit operation. The boolean operations, and ($\wedge$), or ($\vee$), and not ($\sim$) each cost one step. Our fundamental operations are arithmetic operations so we need to find the bit complexity of +, -, *, $\div$, and exponentiation. Exponentiation is explored in Section 5.4, so we skip it for now. Table 1 from the text (pg. 43) gives the cost of the basic arithmetic operations.

2 Languages and Complexity Classes

We would like to quantify how difficult a problem is to solve. Problems with different inputs and outputs are hard to compare, so we restate the problems in such a way that comparing two problems is easier. Our first simplification is to restrict the possible answers for a problem. A problem is a

decision problem if the only answers to the problem are either YES or NO. Decision problems are usually presented with a name, an instance, and a question to answer.

Example 2.1. **Problem:** PRIMES
Instance: A non-negative integer n.
Question: Is n a prime number?

Example 2.2. **Problem:** EVEN NUMBERS
Instance: An integer n.
Question: Is n an even integer?

Example 2.3. **Problem:** SATISFIABILITY
Instance: A set U of boolean variables and a collection C of boolean clauses over U.
Question: Is there a truth assignment of the variables in U such that every clause in C is true?

The second simplification is to define how input is encoded for an algorithm to solve a decision problem. An **alphabet**, denoted Σ, is a set of symbols used to encode input. For this course, we will use the alphabet $\Sigma = \{0, 1, \#\}$. A **string** is a sequence of symbols from Σ. Examples of strings are 0110 and ##01#1#0. The set of all strings is $\Sigma^* = \{\lambda, 0, 1, \#, 00, 01, 0\#, 10, 11, 1\#, \ldots\}$, where λ represents the empty string. The **length** of a string $x \in \Sigma^*$ is $\lg x$, also denoted as $|x|$. A **language** L is any subset $L \subseteq \Sigma^*$. We can call the set of all instances that have a YES answer for a given decision problem DP the language for DP.

Example 2.4. $\mathbb{N}$ is the set of instances for PRIMES. We can encode the instances in binary form as $\{0, 1, 10, 11, 100, \ldots\}$. The language of all instances that give a YES answer to PRIMES is

$$\text{PRIMES} \quad = \quad \{10, 11, 101, ...\}$$

In this case, PRIMES is the language for PRIMES.

An algorithm that takes a string x as input and returns 1 if $x \in L$ and 0 otherwise **accepts** L. We would like to classify languages and the algorithms that accept them. A set of languages is called a **complexity class**. Placing languages into complexity classes is a way to help us understand how difficult problems are.

2.1 Worst Case Complexity

The complexity of most interest in this course is **time complexity**, the number of steps an algorithm takes before it halts. If A is an algorithm, we define $T(A, x)$ to be the number of bit steps A takes on input x. The **worst case time complexity** $T_A(n)$ is the maximum number of steps A uses for an input of length n; that is,

$$T_A(n) = \max_{n=\lg x} T(A, x).$$

Let A, B, C be three different algorithms to solve the same decision problem. These three algorithms might have worst case time complexities $T_A(n) = O(n)$ (i.e., linear time), $T_B(n) = O(n^2)$ (i.e.,

quadratic time), and $T_C(n) = O(n \log n)$. Since we are using O, we cannot compare A, B, and C definitely, although A is most likely to be the best algorithm to use in the worst case. If instead of O, Θ had been used, we could conclude that $A < C < B$ in the worst case.

A deterministic algorithm, A, **computes in polynomial time** if there is a polynomial, p, such that $T_A(n) = O(p(n))$ (Definition 3.3.1, pg. 45).

Example 2.5. Let ADDITION $= \{0\#0\#0, 0\#1\#1, 1\#0\#1, 1\#1\#10, 10\#1\#11, ...\}$, using $\#$ as a separator. So $0\#0\#0$ means $0 + 0 = 0$. A YES instance for this problem looks like $x = m\#n\#(m+n)$. The length of an instance is $\lg x = \lg m + \lg n + \lg(m+n) + 2$. Suppose algorithm A recognizes ADDITION. Then $T_A(\lg x) = \lg m + \lg n + \lg(m+n) + C$, where C is constant overhead. We can asymptotically bound the worst case time complexity by choosing $p(y) = 2y$. Then $T_A(\lg x) \leq p(\lg x)$.

The first complexity class is P.

$$P = \{L | L \text{ is accepted by some deterministic polynomial time algorithm}\}.$$

P corresponds to efficiently (deterministically) solvable problems.

A **nondeterministic algorithm** is one in which choices are made nondeterministically, or by guessing. If there is some collection of guesses for which the algorithm answers YES for input x, then the algorithm accepts x (pg. 46).

Example 2.6. Let COMPOSITES $= \{100, 110, 1000, 1001, ...\}$. A nondeterministic algorithm takes input x and guesses two integers y and z. If $x = yz, y > 1$, and $z > 1$, then the algorithm accepts x.

The second complexity class is NP, defined on pg. 46 in the text.

$$NP = \{L | \text{there is a language } S \in P \text{ and a polynomial p such that } x \in L \text{ if and only if there is some string y such that } \lg y \leq p(\lg x) and (x, y) \in S\}.$$

Clearly $P \subseteq NP$. However, it is not known whether $P \neq NP$.

The complexity class $co - NP$ is obtained by complementing the members of NP.

Example 2.7. COMPOSITES $\in NP$, so PRIMES $\in co - NP$. Note that it also turns out that PRIMES $\in NP$.